March 13, 1959

Artificial Intelligence Project---RLE and MIT Computation Center

Memo 8

RECURSIVE FUNCTIONS OF SYMBOLIC EXPRESSIONS AND THEIR COMPUTATION BY MACHINE

by J. McCarthy

An Error in Memo 8

The definition of eval given on page 15 has two errors, one of which is typographical and the other conceptual. The typographical error is in the definition of evcon where "1→" and "T→" should be interchanged.

The second error is in evlam. The program as it stands will not work if a quoted expression contains a symbol which also acts as a variable bound by the lambda. This can be corrected by using instead of subst in evlam a function subsq defined by

```
subsq=λ[[x;y;z];[null[z]→Λ;atom[z]→
[y=z→x;1→z];first[z]=QUOTE→z; 1→
combine[subsq[x;y;first[z]];subsq[x;y;rest[z]]]]]
```

March 4, 1959

Artificial Intelligence Project---RLE and MIT Computation Center

Memo 8

RECURSIVE FUNCTIONS OF SYMBOLIC EXPRESSIONS AND THEIR COMPUTATION BY MACHINE

by J. McCarthy

The attached paper is a description of the LISP system starting with the machine-independent system of recursive functions of symbolic expressions. This seems to be a better point of view for looking at the system than the original programming approach. After revision, the paper will be submitted for publication in a logic or computing journal.

This memorandum contains only the machine independent parts of the system. The representation of S-expressions in the computer and the system for representing S-functions by computer subroutines will be added.

RECURSIVE FUNCTIONS OF SYMBOLIC EXPRESSIONS AND THEIR COMPUTATION BY MACHINE

by John McCarthy, MIT Computation Center

1. Introduction

A programming system called LISP (for LIS Processor) has been developed for the IBM 704 computer by the Artificial Intelligence Group at MIT. The system was designed to facilitate experiments with a proposed system called the Advice Taker whereby a machine could be instructed in declarative as well as imperative sentences and could exhibit "common sense" in carrying out its instructions. The original proposal for the Advice Taker is contained in reference 1. The main requirement was a programming system for manipulating expressions representing formalized declarative and imperative sentences so that the Advice Taker system could make deductions.

The development of the LISP system went through several stages of simplification in the course of its development and was eventually seen to be based on a scheme for representing the partial recursive functions of a certain class of symbolic expressions. This representation is independent of the IBM 704 or any other electronic computer and it now seems expedient to expound the system starting with the class of expressions called S-expressions and the functions called S-functions.

In this paper, we first describe the class of S-expressions and S-functions. Then we describe the representation of S-functions by S-expressions which enables us to prove that all computable partial functions have been obtained, to obtain a universal S-function, and to exhibit a set of questions about S-expressions which cannot be decided by an S-function. We describe the representation of the system in the IBM 704, including the representation of S-expressions by list structures similar to those used by Newell, Simon, and Shaw (see reference 2), and the representation of S-functions by subroutines. Finally, we give some applications to symbolic calculations including analytic differentiation, proof checking, and compiling including a description of the present status of the LISP compiler itself which is being written in the system.

Although we have not carried out the development of

recursive function theory in terms of S-functions and their representation by S-expressions beyond the simplest theorems, it seems that formulation of this theory in terms of S-functions has important advantages. Devices such as Gödel numbering are unnecessary and so is the construction of particular Turing machines. (These constructions are all artificial in terms of what is intended to be accomplished by them). The advantage stems from the fact that functions of symbolic expressions are easily and briefly described as S-expressions and the representation of S-functions by S-expressions is trivial. Moreover, in a large class of cases the S-expression representations of S-functions translate directly into efficient machine programs for the computation of the functions. Although, the functions described in the manner of this paper include all computable functions of S-expressions, describe many important processes in a very convenient way, and compile into fast running programs for carrying out the processes; there are other kinds of processes whose description by S-functions is inconvenient and for which the S-functions once found do not naturally compile into efficient programs. For this reason, the LISP system includes the possibility of combining S-functions into Fortran or IAL-like programs. Even this will not provide the flexibility of description of processes hoped for from the Advice Taker system which is beyond the scope of this paper.

2. Recursive Functions of Symbolic Expressions

In this section we define the S-expressions and the S-functions. (Actually they are partial functions. The distinction between a partial function and a function that the former need not be defined for all arguments because, for example, the computation process defining it may not terminate.)

2.1. S-expressions

The expression with which we shall deal are formed using the special characters "," and "(" and ")" and an infinite set of distinguishable atomic symbols $p_1, p_2, p_3, \ldots$

The S-expressions are formed according to the following recursive rules.

1. The atomic symbols p_1 p_2 etc. are S-expressions.
2. A null expression Λ is also admitted.
3. If e is an S-expression so is (e).
4. If e_1 and (e_2) are S-expressions so is (e_1,e_2).

In what follows we shall use sequences of capital Latin letters as atomic symbols. Since we never juxtapose them without intervening commas they cannot cause confusion by running together. Some examples of S-expressions are:

AB
(AB,A)
(AB,A,,C,)
((AB,C),A,(BC,(B,B)))

2.2 Elementary functions and predicates.

The functions we shall need are built up from certain elementary ones according to certain recursive rules.

There are three elementary predicates:

1. null[e]

null[e] is true if and only if S-expression e is the null expression Λ. (We shall use square brackets and semi-colons for writing functions of S-expressions since parentheses and commas have been pre-empted. When writing about functions in general we may continue to use parentheses and commas.)

2. atom[e]

atom[e] is true if and only if the S-expression is an atomic symbol.

3. $p_1=p_2$

$p_1=p_2$ is defined only when p_1 and p_2 are both atomic symbols in which case it is true if and only if they are the same symbol. This predicate expresses the distinguishability of the symbols.

There are three basic functions of S-expressions whose values are S-expressions.

4. first[e]

first[e] is defined for S-expressions which are neither null nor atomic. If e has the form (e_1,e_2) where e_1 is an expression, then first[e] $=e_1$. If e has the form (e_1) where e_1 is an S-expression again we have first[e] $=e_1$.

Some examples are:

first[(A,B)] =A

first[A] is undefined

first[(A)] =A

first[((A,B),C,D)] =(A,B)

5. rest[e]

rest[e] is also defined for S-expressions which are neither null nor atomic. If e has the form (e_1,e_2) where e_1 is an S-expression, then rest[e] $=(e_2)$. If e has the form (e_1) where e_1 is an S-expression we have rest[e] $=\Lambda$.

Some examples are:

rest[(A,B)] =(B)

rest[(A)] $=\Lambda$

rest[(A,B,C)] =(B,C)

6. combine[e_1;e_2]

combine[e_1;e_2] is defined when e_2 is not atomic.

When e_2 has the form (e_3), then combine[e_1;e_2] $=(e_1,e_3)$

When e_2 is Λ we have combine[e_1;Λ] $=(e_1)$.

Some examples are:

combine[A;Λ] $=(A)$

combine[(A,B);(B,C)] $=((A,B),B,C)$

The functions first, rest and combine are related by the relations

first[combine[e_1;e_2]] $=e_1$

rest[combine[e_1;e_2]] $=e_2$

combine[first[e];rest[e]] $=e$

whenever all the quantities involved are defined.

2.3 Functional Expressions and Functions formed from the elementary functions by composition.

Additional functions may be obtained by composing the elementary functions of the preceding section. These functions are described by expressions in the meta-language which should not be confused with the S-expressions being manipulated. For example, the expression first[rest[e]] denotes the second sub-expression of the S-expression e, e.g. first[rest[(A,B,C)]] $=B$. In general compositions of first and rest give sub-expressions of an S-expression in a given position within the expression and

compositions of combine form S-expressions from their sub-expressions. For example, combine [x;combine[y;combine[z,Λ]]] forms a sequence of three terms from the terms, e.g. combine [A;combine[(B,C);combine[A,Λ]]]=(A,(B,C),A).

In order to be able to name compositions of functions and not merely functional expressions (forms) we use the Church λ-notation. If $\mathcal{E}$ is a functional expression and $x_1,\ldots,x_n$ are variables which may occur in $\mathcal{E}$, then λ[[$x_1,\ldots,x_n$],] denotes the function of n variables that maps $x_1,\ldots,x_n$ into $\mathcal{E}$. For example, λ[[x],first[rest[x]]]is the function which selects the second element of a list and we have λ[[x];first[rest[x]]][(A,[B,C],A)]=[B,C]. λ[[x];[A,B]]is the constant function that maps every S-expression into [A,B].

The variables occurring in the list of a λ-expression are bound and replacing such a variable throughout a λ-expression by a new variable does not change the function represented. Thus λ[[x,y], combine[x,combine[y,Λ]]]is the same function as λ[[u,v], combine[u,combine[v,Λ]]]but different from λ[[y,x], combine[x,combine[y,Λ]]].

If some of the variables in a functional expression or form are bound by λ's and others are not, we get a function dependent on parameters or from another point of view a form whose value is a function when values have been assigned to the variables.

2.4 Conditional Expressions

Let $p_1,p_2,\ldots,p_k$ be expressions representing propositions and let $e_1,\ldots,e_k$ be arbitrary expressions. The expression $[p_1\rightarrow e_1,\ldots p_k\rightarrow e_k]$ is called a conditional expression and its value is determined from the values assigned to the variables occurring in it as follows: If the value of p_1 is not defined neither is that of the conditional expression. If p_1 is defined and true the value of the conditional expression is that of e_1 if the latter is defined and otherwise is undefined. If p_1 is defined and false, then the value of $[p_1\rightarrow e_1,\ldots,p_k\rightarrow e_k]$ is that of $[p_2\rightarrow e_2,\ldots p_k\rightarrow e_k]$. Finally if p_k is false the value of $[(p_k\rightarrow e_k)]$ is undefined.

An example of a conditional expression is [null[x]→Λ; atom[x]→Λ;1→first[x]]. The "1" occurring in the above expression is the propositional constant "truth". We also use

"0" for the propositional constant "falsehood". When used as the last proposition in a conditional expression "1" may be read "in all remaining cases". The expression given is a sort of extension of the expression first[x] which is defined for all S-expressions. We could define a corresponding function by first a=λ[[x];[null[x]→A;atom[x]→A;1→first[x]]].

It is very important to note that for a conditional expression to be defined it is not necessary for all of its sub-expressions to be defined. If p_1 is defined and true and e_1 is defined, the conditional expression $[p_1 \rightarrow e_1, \ldots p_k \rightarrow e_k]$ is defined even if none of the other p's or e's is defined. If p_1 is defined and false, p_2 is defined and true and e_2 is defined, the expression is defined even if e_1 and all the other p's and e's are undefined.

The propositional connectives $\wedge$ and $\vee$ and $\sim$ may be defined in terms of conditional expressions. We have $p_1 \wedge p_2 = [p_1 \rightarrow [p_2 \rightarrow 1, 1 \rightarrow 0], 1 \rightarrow 0]$ and $p_1 \vee p_2 = [p_1 \rightarrow 1, p_2 \rightarrow 1, 1 \rightarrow 0]$ and $\sim p = [p \rightarrow 0, 1 \rightarrow 1]$ There is a slight difference between the connectives defined this way and the ordinary connectives. Suppose that p_1 is defined and true but p_2 is undefined. Then $p_1 \vee p_2$ is defined and true but $p_2 \vee p_1$ is undefined.

2.5 Recursive Function Definitions

The functions which can be obtained from the elementary functions and predicates by composition and conditional expressions form a limited class. As we have described them they are not defined for all S-expressions but if we modified the definitions of the elementary functions so that the undefined cases are defined in some trivial way, as in the example of the previous section, they would be always defined.

Additional functions may be defined by writing definitions of the form,

$f = \lambda[[x_1 \ldots x_n], \mathcal{E}]$ where the expression $\mathcal{E}$ may contain the symbol f itself. A function f defined in this way is to be computed for a given argument is to be computed by substitution of the argument into the expression and attempting to evaluate the resulting expression. When a conditional expression is encountered we evaluate p's until we find a true p and then evaluate the corresponding e. No attempt is made

to evaluate later p's or any e except the one corresponding to the first e. It may happen that in evaluating for given values of the variables it is unnecessary to evaluate any expression involving the defined function f. In this case, the evaluation may be completed and the function defined for this argument. If expressions involving f do have to be evaluated we substitute the arguments of f and again proceed to evaluate. The process may or may not terminate. For those arguments for which the process does terminate the function is defined.

We shall illustrate this concept by several examples:

1. Our first example is a function which gives the first symbol of an expression.

We define

ff=λ[[x];[null[x]∨atom[x]→x;1→ff[first[x]]]]

Let us trace the computation of ff[(((A),B),C)]. We have

ff[(((A),B),C)]=[null[(((A),B),C)]∨atom[(((A),B,C)]→(((A),B),C);
1→ff[first[(((A),B),C)]]
=ff[((A),B)]
=[null[((A),B)]∨atom[((A),B)]→((A),B);1→ff[first[((A),B)]]]
=ff[(A)]
=[null[(A)]∨atom[(A)]→(A);1→ff[first[(A)]]]
=ff[A]
=[null[A]∨atom[A]→A;1→ff[first[A]]]
=A

* Note that it does not matter that first A occurring in the next to last step is undefined.

2. The second example is a function which gives the result of substituting the expression x for the symbol y in the expression s. We define

subst=λ[[x;y;s];[null[s]→Λ;atom[s]→[y=s→x;1→s];1→
combine[subst[x;y;first[s]];subst[x;y;rest[s]]]]]

We shall illustrate the application of this definition by computing subst[(A,B);X;((X,A),C)]. In order to make the tracing shorter we shall give the situation at each recursion and leave it to the reader to substitute the definition of each subst expression and to check the determination of which case of the

conditional is applicable. We have

subst[(A,B);X;(X,A),C)]=

=combine[subst[(A,B);X;(X,A)];subst[(A,B);X;(C)]]

=combine[combine[subst[(A,B);X;X];subst[(A,B);X;(A)]];combine[

subst[(A,B);X;C];subst[(A,B);X;Λ]]]

=combine[combine[(A,B);combine[subst[(A,B);X;A];subst[(A,B)

;X;Λ]]];combine[C;Λ]]

=combine[combine[(A,B);combine[A;Λ]];(C)]

=(((A,B),A),C)

2.6 Functions with Functions as Arguments

If we allow variables representing functions to occur in expressions and create functions by incorporating these variables as arguments of λ's we can define certain functions more concisely than without this facility. However, as we shall show later no additional S-functions become definable.

As an example of this facility we define a function maplist [x,f] where x is an S-expression and f is a function from S-expressions to S-expressions. We have

maplist=λ[[x,f];[null[x]→Λ;1→combine[f[x];maplist[rest[x];f]]]]

The usefulness of maplist is illustrated by formulas for the partial derivative with respect to x of expressions involving sums and products of x and other variables. The S-expressions we shall differentiate are formed as follows:

1. An atomic symbol is an allowed expression.
2. If $e_1;e_2;\ldots;e_n$ are allowed expressions so are (PLUS,e_1, $\ldots,e_n$) and (TIMES,$e_1,\ldots,e_n$) and represent the sum and product respectively of $e_1;\ldots;e_n$.

This is essentially the Polish notation for functions except that the inclusion of parentheses and commas allows functions of variable numbers of arguments. An example of an allowed expression is

(TIMES, X,(PLUS, X,A),Y)

the conventional algebraic notation for which is X(X+A)Y

Our differentiation formula is

diff=λ[[y,x];[atom[y]→[y=x→ONE;1→ZERO];

first[y]=PLUS→combine[PLUS;maplist[rest[y];λ[[s];diff[

first[s];x]]]];first[y]=TIMES→combine[PLUS;maplist[

rest[y];λ[[s];combine[TIMES;maplist[rest[y];λ[w];[s≠w

→first[w];1→diff[first[w];x]]]]]]]]]

The derivative of the above expression computed by this formula is

(PLUS,(TIMES,ONE,(PLUS,X,A),Y),(TIMES,X,(PLUS,ONE,ZERO),Y),
(TIMES,X,(PLUS,X,A),ZERO))

2.7 Labelled Expressions

The λ-notation used for naming functions is inadequate for naming recursive functions. For example, if the function named as the second argument of a maplist is to be allowed to be recursive an additional notation is required.

We define label[a;e] where a is a symbol and e is an expression to be the same as the expression e except that if a occurs as a sub-expression of e it is understood to refer to the expression e. The symbol a is bound in label [a;e] and has no significance outside this expression. Label acts as a quantifier with respect to its first argument but a quantifier of a different sort from λ. As an example

```
label[subst;λ[[x;y;s];[null[s]→Λ;atom[s]→
    [y=s→x;1→s];1→combine[subst[x;y;first[s]];
        subst[x;y;rest[s]]]]]]
```

is a name suitable for inclusion in a maplist of the substitution function mentioned earlier.

2.8 Computable Functions

In this section we shall show that all functions computable by Turing machine are expressable as S-functions. If, as we contend, S-functions are a more suitable device for developing a theory of computability than Turing machines, the proof in this section is out of place and should be replaced by a plausibility argument similar to what is called "Turing's thesis" to the effect that S-functions satisfy our intuitive notion of effectively computable function. The reader unfamiliar with Turing machines should skip this section.

Nevertheless, Turing machines are well entrenched at present so we shall content ourselves with showing that any function computable by Turing machine is an S-function. This is done as follows:

1. We give a way of describing the instantaneous configurations of a Turing machine calculation by an S-expression. This S-expression must describe the Turing machine, its

internal state, the tape, and the square on the tape being read.

2. We give an S-function succ whose argument is an instantaneous configuration and whose value is the immediately succeeding configuration if there is one and otherwise is 0.

3. We construct from succ another S-function, turing whose arguments are a Turing machine, with a canonical initial state and an initial tape in a standard position and whose value is defined exactly when the corresponding Turing machine calculation terminates and in that case is the final tape.

We shall consider Turing machines as given by sets of quintuples. Each quintuple consists of a state, a symbol read, a symbol to be printed, a direction of motion and a new state. The states are represented by a finite set of symbols, the symbols which may occur by another finite set of symbols (it doesn't matter whether these sets overlap) and the two directions by the symbols "L" and "R". A quintuple is then represented by an S-expression (st,sy,nsy,dir,nst). The Turing machine is represented by an S-expression. (ist, blank,quin1,...,quink) where ist represents the canonical initial state, blank is the symbol used for a blank square (squares beyond the region explicitly represented in the S-expression for a tape are assumed to be blank and are read that way when reached). As an example we give the representation of a Turing machine which moves to the right on a tape and computes the parity of the number of 1's on the tape ignoring 0's and stopping when it comes to a blank square:
(0,B,(0,0,B,R,0),(0,1,B,R,1),(0,B,0,R,2),(1,0,B,R,1),(1,1,B,R,0),
(1,B,1,R,2))
The machine is assumed to stop if there is no quintuple with a given symbol state pair so that the above machine stops as soon as it enters state 2.

A Turing machine tape is represented by an S-expression as follows: The symbols on the squares to the right of the scanned square are given in a list v, the symbols to the left of the scanned square in a list u and the scanned symbol as a quantity w. These are combined in a list (w,u,v). Thus the tape ...bb1101011Ob.... is represented by the expression

(0,(1,0,1,1,b,b),(1,1,0,b))

We adjoin the state to this triplet to make a quadruplet (s,w,u,v) which describes the instantaneous configuration of a machine.

The function succ[m;c] whose arguments are a Turing machine m and a configuration c has as value the immediately succeeding configuration of c provided the state-symbol pair is listed among the quintuplets of m and otherwise has value zero.

succ is defined with the aid of auxiliary functions. The first of these find[st;sy;qs] gives the triplet (nsy;dir;nst) which consist of the last 3 terms of the quintuplet of m which contains (st,sy) as its first two elements. The recursive definition is simplified by defining find [st;sy;qs] where qs=rest[rest[m]] since qs then represents the list of quintuplets of m. We have find[st;sy;qs]=[null[qs]→0;first[first[qs]]=st∧first[rest[first[qs]]]=sy→rest[rest[first[qs]]];1→find[st;sy,rest[qs]]]

The new auxiliary function is move[m;nsy;tape;dir] which gives a new tape triplet obtained by writing nsy on the scanned square of tape, and moving in the direction dir.

move[m,nsy;tape;dir]=[dir=L→combine[[null[first[rest[tape]]]→first[rest[m]];1→first[first[rest[tape]]]];combine[[null[first[rest[tape]]]→Λ;1→rest[first[rest[tape]]]];combine[combine[nsy;first[rest[rest[tape]]];Λ]]];dir=R→combine[[null[first[rest[rest[tape]]]]→first[rest[m]];1→first[first[rest[rest[tape]]]]];combine[combine[nsy;first[rest[tape]]];combine[[null[first[rest[rest[tape]]]]→Λ;1→rest[first[rest[rest[tape]]]];Λ]]]]

The reader should not be alarmed at the monstrous size of the last formula. It rises mainly from the compositions of first and rest required to select the proper elements of the structure representing the tape. Later we shall describe ways of writing such expressions more concisely.

We now have

succ[m;c]=[find[first[c];first[rest[c]];rest[rest[m]]]=0→0;1→combine[first[rest[rest[find[first[c];first[rest[c]];rest[rest[m]]]]]];move[m;first[find[first[c];first[rest[c]];rest[rest[m]]]];rest[c];first[rest[find[

Finally we define

turing[m;tape] = tu[m;combine[first[m];tape]]

where

tu[m;c] = [succ[m;c]=0→rest[c];1→tu[m;succ[m;c]]]

We reiterate that these definitions can be greatly shortened by some devices that will be discussed in the sections on the machines computation of S-functions.

3. Lisp Self-applied

The S-functions have been described by a class of expressions which has been informally introduced. Let us call these expressions F-expressions. If we provide a way of translating F-expressions into S-expressions, we can use S-functions to represent certain functions and predicates of S-expressions.

First we shall describe this translation.

3.1 Representation of S-functions as S-expressions.

The representation is determined by the following rules.

1. Constant S-expressions can occur as parts of the F-expressions representing S-functions. An S-expression $\mathcal{E}$ is represented by the S-expression. (QUOTE,$\mathcal{E}$)

2. Variables and function names which were represented by strings of lower case letters are represented by the corresponding strings of the corresponding upper case letters. Thus we have FIRST, REST and COMBINE, and we shall use X,Y etc. for variables.

3. A form is represented by an S-expression whose first term is the name of the main function and whose remaining terms are the arguments of the function. Thus combine[first[x]; rest[x]] is represented by (COMBINE,(FIRST,X),(REST,X))

4. The null S-expression Λ is named NIL.

5. The truth values 1 and 0 are denoted by T and F. The conditional expression

$$\text{write}[p_1 \rightarrow e_1;\ p_2 \rightarrow e_2;\ \ldots,\ p_k \rightarrow e_k]$$

is represented by

$$(\text{COND},(p_1,e_1),(p_2,e_2),\ldots,(p_k,e_k))$$

6. $\lambda[[x;\ldots;z];\mathcal{E}]$ is represented by (LAMBDA,(X,...,Z);$\mathcal{E}$)

7. label[$\mathcal{A}$;$\mathcal{E}$] is represented by (LABEL,$\mathcal{A}$,$\mathcal{E}$)

8. x=y is represented by (EQ,X,Y)

With these conventions the substitution function mentioned earlier whose F-expression is

```
label[subst;λ[[x;y;s];[null[s]→Λ;atom[s]→
    [y=s→x;1→s];1→combine[subst[x;y;first[s]];
        subst[x;y;rest[s]]]]]]
```

is represented by the S-expression.

```
(LABEL,SUBST,(LAMBDA,(X,Y,Z),(COND,((NULL,
    Z),NIL),((ATOM,Z),(COND,((EQ,Y,Z),X),(1,Z))),
        (1,(COMBINE,(SUBST,X,Y,(FIRST,Z)),
            (SUBST,X,Y,(REST,Z))))))))
```

This notation is rather formidable for a human to read, and when we come to the computer form of the system we will see how it can be made easier by adding some features to the read and print routines without changing the internal computation processes.

3.2. A Function of S-expressions which is not an S-function.

It was mentioned in section 2.5 that an S-function is not defined for values of its arguments for which the process of evaluation does not terminate. It is easy to give examples of S-functions which are defined for all arguments, or examples which are defined for no arguments, or examples which are defined for some arguments. It would be nice to be able to determine whether a given S-function is defined for given arguments. Consider, then, the function def[f;s] whose value is 1 if the S-function whose corresponding S-expression is f is defined for the list of arguments s and is zero otherwise.

We assert that def[f; s] is not an S-function. (If we assume Turing machine theory this is an obvious consequence of the results of section 2.8, but in support of the contentions that S-functions are a good vehicle for expounding the theory of recursive functions we give a separate proof).

Theorem: def[f;s] is not an S-function.

Proof: Suppose the contrary. Consider the function

$g=\lambda[[f];[\sim def[f;f]\rightarrow 1, 1\rightarrow first[\Lambda]]]$

If def were an S-function g would also be ab S-function. For any S-function u with S-expression u' g[u'] is 1 if u[u'] undefined and is undefined otherwise.

Consider now g[g'] where g' is an S-expression for g. Assume first that g[g'] were defined. This is precisely the condition that g' be the kind of S-expression for which g is undefined. Contrawise, were g[g'] undefined g' would be the kind of S-expression for which g is defined.

Thus our assumption that def[f;s] is an S-function leads to a contradiction.

The proof is the same as the corresponding proof in Turing machine theory. The simplicity of the rules by which S-functions are represented as S-expressions makes the development from scratch simpler, however.

3.3 The Universal S-Function, Apply

There is an S-function apply such that if f is an S-expression for an S-function and args is a list of the form (arg1,...,arg n) where arg1,---,arg n are arbitrary S-expressions then apply[f,args] and φ[arg1;...;arg n] are defined for the same values of arg1,...,arg n and are equal when defined.

apply is defined by

apply[f;args] = eval[combine[f;args]]

eval is defined by

```
eval[e] = [
first[e] = NULL → [null[eval[first[rest[e]]]] → T; 1 → F]
first[e] = ATOM → [atom[eval[first[rest[e]]]] → T; 1 → F]
first[e] = EQ → eval[first[rest[e]]] = eval[first[rest[rest[e]]]] → T;
      1 → F]
first[e] = QUOTE → first[rest[e]];
first[e] = FIRST → first[eval[first[rest[e]]]];
first[e] = REST → rest[eval[first[rest[e]]]];
first[e] = COMBINE → combine[eval[first[rest[e]]];eval[first[rest[rest
      [e]]]]];
first[e] = COND → evcon[rest[e]];
first[first[e]] = LAMBDA → evlam[first[rest[first[e]]];first[rest[rest
      [first[e]]]];rest[e]];
first[first[e]] = LABEL → eval[combine[subst[first[e];first[rest
      [first[e]]];first[rest[rest[first[e]]]];rest[e]]]]
where; evcon[c] = [eval[first[first[c]]] = 1 → eval[first[rest[first[c]]]];
            T → evcon[rest[c]]]
and
evlam[vars;exp;args] = [null[vars] → eval[exp];1 → evlam[
      rest[vars];subst[first[args];first[vars];exp];rest[args]]]
```

The proof of the above assertion is by induction on the subexpressions of e. The process described by the above functions is exactly the process used in the hand-worked examples of section 2.5.

4. Variants of Lisp

There are a number of ways of defining functions of symbolic expressions which are quite similar to the system we have adopted. Each of them involves three basic functions, conditional expressions and recursive function definitions, but the class of expressions corresponding to S-expressions differs and so do the precise definitions of the functions. We shall describe two of these variants.

4.1 Linear Lisp

The L-expressions are defined as follows:

1. A finite list of characters is admitted.

2. Any string of admitted characters is an L-expression. This includes the null string denoted by Λ

There are three functions of strings

1. first[x] is the first character of the string x.

   first[Λ] is undefined.

For example, first [ABC]=A.

2. rest[x] is the string of characters remaining when the first character of the string is deleted.

   rest[Λ]is undefined.

   For example, rest[ABC]=BC

3. combine[x;y] is the string formed by prefixing the character x to the string y.

   For example, combine[A;BC]=ABC

There are three predicates on strings

1. char[x], x is a single character
2. null[x], x is the null string
3. x=y, defined for x and y characters.

The advantage of linear Lisp is that no characters are given special roles as are parentheses and comma in Lisp. This permits computations with any notation which can be written linearly. The disadvantage of linear Lisp is that the extraction of sub-expressions is a fairly involved rather than an elementary operation. It is not hard to write in linear Lisp functions corresponding to the basic functions of Lisp so that mathematically linear Lisp includes Lisp. This turns out to be the most convenient way of programming more complicated manipulations. However, it turns out that if the functions are to be represented by computer routines Lisp is essentially faster

4.2 Binary Lisp

The unsymmetrical status of first and rest may be a source of uneasiness. If we admit only two element lists then we can define

$$\text{first}[(e_1,e_2)]=e_1$$
$$\text{rest}[(e_1,e_2)]=e_2$$
$$\text{combine}[e_1;e_2]=(e_1,e_2)$$

We need only two predicates, equality for symbols and atom. The null list can be dispensed with. This system is easier until we try to represent functions by expressions which is, after all, the principal application; moreover, in order to apply the system to itself we need to be able to write functions.